ICAS-98-1,9,2

A98-31476

ALLOCATION OF FAULT HANDLING TECHNIQUES IN MULTIPROCESSING AVIONICS ARCHITECTURES

A. Marchetto
Alenia Aeronautica
Stabilimento Caselle Nord
Strada Privata, 10072 Caselle Torinese, Italy

Abstract

Future avionics architectures will be characterized by the implementation of distributed processing at ever increasing levels. A modular approach to design offers the suitable physical structure to accommodate this trend, and, at the same time, promises great improvements of operational and mission performances. In particular, an high tolerance to faults is expected, as most avionics functions become software functions in the computing core, competing for modular reconfigurable shared resources to accomplish their tasks.

This paper origins from research activities carried out by Alenia, at both European and National level, in the area of integrated modular avionics architectures, and develops the following subjects:

- Differentiation and definition of general fault-preventing methods for avionics system: fault avoidance, fault removal, fault tolerance.
- Definition of the mix of techniques which will co-operate in providing the required degree of fault tolerance.
- Identification of the categories of fault which have to be addressed at distributed operating system level, and characterization of the services which have to be provided in support of the above mix of fault tolerance techniques.

Finally, the paper outlines the physical architecture of a general modular multiprocessing core and shows how the application of proper mixes of fault tolerance techniques allows recovering of different kinds of faults.

Introduction

Multiprocessing architectures have become a reality for advanced avionics applications. Interesting, in particular, are the dependability improvement obtainable with distributed architectures, that is, the probability that the quality of service required by a particular function will be provided over a defined period of time. Quality of service is a combination of a number of factors as reliability, availability, safety, security, maintainability, testability, accuracy, precision and latency. Important attributes of distributed processing architectures to these respects are [1]:

- Functional integration
- Parallel high performance computation
- Scaleability (the processing power grows virtually transparently as processing nodes are added)
- Selective technology upgrade
- Adjustable levels of functions reliability
- Graceful degradation of system capabilities in the presence of faults

Nevertheless, at a closer look, things are not so straight, since an obvious effect of adding more and more components beyond a certain number is that the possibility of single component fault grows [2], with negative consequences on system reliability. On the other hand modular distributed systems are so flexible from the point of view of fault tolerance techniques that the net result is, when a correct approach to design is applied, improved dependability with respect to centralized systems. The allocation of proper fault management techniques at software architecture level is key factor in achieving the projected benefits [3].

Proposed Software Architecture

International research programmes encompassing integrated modular avionics, such as EUCLID (European Co-operation for Long Term in Defence) CEPA 4 (Common European Priority Area 4 - Modular Avionics) RTP (Research and Technology programme) 4.1, or ASAAC (Allied Standard Avionic Architecture Council) phase 1, have indicated a layered software architecture as the means to fulfil important requirements, such as software portability and reusability, and, in particular, system fault tolerance, which is the main subject of this paper. With reference to Fig. 1 [4], the application layer

consists of functional applications and system applications. Functional applications cover the universe of software applications required to fulfill the specific mission requirements, such as those related to functionality's of Mission Management, Sensors Management, Threat / Target Management, Navigation Management, Crew Interface Management. System applications are responsible for control and management of the functional applications and the hardware resources, supplying services as scheduling, communication management, fault management and configuration / reconfiguration management in compliance to what required by a structured and real-time accessible system description, realized off-line and embedded in a set of "blueprints".

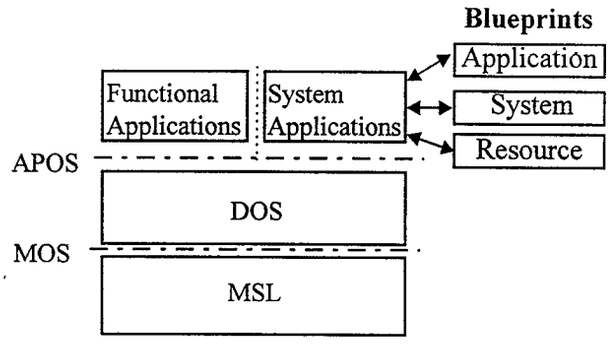


Fig 1 - Software Architecture Model

Three families of blueprints are identified [5] :

- Application blueprints, describing the requirements of the universe of applications
- Resources blueprints, describing the physical system
- System blueprints, matching the virtual system description given by application blueprints to the physical system description given by resources blueprints

Ref[5], focused on the blueprint concept, presents a practical application case.

The operating system layer provides all applications with a set of standardised services through a standardised interface (APOS, Application to Operating System interface). Each module hardware functionality must be supported by software, called MSL (Module Support Layer), providing the operating system with standardised services through a standardised interface (MOS, Module to Operating System interface). The MSL resides on every module and is supplied by the module manufacturer. The operating system resides on every module also, at least the basic functionalities required to control the particular module type. It is therefore a Distributed Operating System (DOS).

Should System Applications disappear from the model, the consequence would be attributing to the DOS not only low-level services, but also capabilities of scheduling, communication manager, configuration manager, fault and reconfiguration manager.

Undesired Events

Undesired events in information processing systems can be classified as follows:

- Faults, which we could subdivide in:
  - Potential faults, due to:
    - external disturbances
    - components defects
    - implementation mistakes
    - specification mistakes
  - Actual faults in the operational system
    - latent defects or supervened alteration of hardware components
    - latent defect of software components
- Errors: alteration of information units of any extension, type or level (wrong sequences of bits in a single data word, time based errors in transmitted information as delays or fail to arrive, alteration of massive data sets as images, etc.)
- Failures: the effects of undesired events at user level

Potential faults can be avoided or removed. Those which are neither avoided nor removed become actual faults in the operational system. It is interesting to note (Fig. 2) the cause-effect relationships between potential faults and actual faults. Actual faults can be tolerated during normal system operation. If not, they become errors in the operational system, which, if not recovered, cause failures at user level. For example, let us consider the simple case of a memory element which is part of a data processing module executing route computation tasks. Implementation mistakes on this element, if not avoided or removed, transition as latent hardware faults to the operational system. These, once that particular memory element is involved in computation tasks, may cause errors in the data words defining latitudes and longitudes of the desired route. These errors may cause wrong indications to be displayed to the pilot, that is, failure of the steering function.

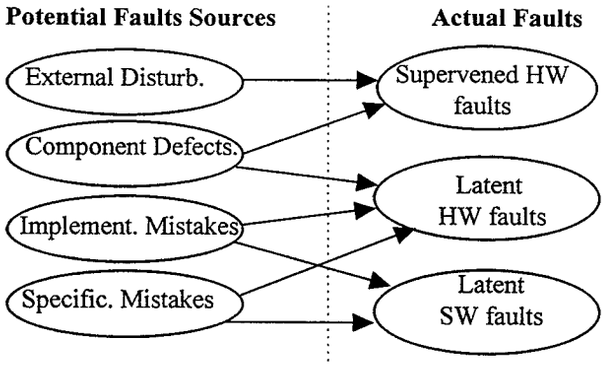


Fig. 2 - Potential Faults and Actual Faults

Avoidance and removal of potential faults

Avoidance methods apply during the development process, reducing:

- Specification and implementation mistakes, using proper development process / methods / tools
- Components defects, by parts screening and acceptance tests
- External disturbances, by shielding, protective structures, dampers, etc.

Removal applies during testing phases, when faults are discovered (due either to specification / implementation mistakes, components defects or external disturbances) which have been introduced in the system during the development phase. Proper actions are then carried out in order to remove the fault from the system, such as

- Modifications of equipment / software / system specifications
- Modifications of hardware / software code / cabling etc. (either because not compliant to specifications or as a result of modifications of specifications)
- Substitution of faulty components
- Introduction of additional precautions against external disturbances.

Fault Tolerance

Tolerance applies to faults which have survived to the testing phase, and become therefore actual faults for the operational system.

As far as hardware components are concerned, faults are often not present since the beginning of the system operation, arising from extended use in hostile environment, and cannot therefore be identified by initial testing or acceptance procedures. On the contrary, software faults are present from the beginning, and, when not detected by test phases, survive latent in the processing system. As a matter of fact, testing is one of the most critical and difficult activities undertaken during software development and maintenance. Proving to a high degree of reliability that complex software applications will execute correctly for all possible inputs is prohibitively expansive and, depending on the required confidence level, may not even be feasible.

We use therefore the term "fault tolerance" to indicate the ability of the system to continue operation in the presence of hardware or software faults. Fault tolerance, opposite to fault avoidance or removal, is strictly dependent on software architecture and on the operative system, relaying on functions as configuration / reconfiguration management, fault management, communication management, scheduling. In the software architecture model presented, these functions pertain to the system application layer, which will manage as examined in the following all activities related to the accomplishment of these functions in accordance to the dynamic system description codified in blueprints. Should system applications not be explicited, these functions would be directly attributed to the operating system, which would need direct access to blueprints.

Techniques for fault tolerant operation

The techniques used to obtain fault tolerant operation, normally referenced as fault detection, identification and recovery (FDIR) methodologies, are the primary means to acquire increased reliability and availability of future multiprocessing modular avionics architectures. They are mainly applied to the handling of hardware faults, being software faults treated with other methods.

In principle, we can distinguish a local level and a system level FDIR [6]. Local level FDIR applied to modular avionics architectures could be implemented at module level, in principle with low operating system involvement. For instance, one of the n processors hosted on a module could be left unloaded as a default procedure, ready to substitute any local processor incurred into a fault. The FDIR mechanisms should be embodied in the module by the supplier, and their operation should tend to be transparent to the DOS. From the system level viewpoint, therefore, we could speak of fault masking, in the sense that virtually no operating system involvement is required to tolerate the fault.

System level FDIR, on the contrary, is not system-transparent (although it should tend to be user-transparent), being carried out under the control of the

DOS and/or the system applications. The FDIR treated in the following is, as correctly stated, system level FDIR. The FDIR concept revolves around the provision of a pool of spare resources which can be used to replace failed components of the same type anywhere in the system within a permitted boundary, delimiting a so called reconfiguration area. These resources are dynamically allocated depending on the specific exigency of the moment, that is, which fault has occurred, where, and what is the current system state and health condition. Once the fault has been detected and identified, an intelligent recovery action has to take place, reconfiguring hardware resources and functional applications, in compliance to what required by the structured dynamic system description given by blueprints. In the following, the FDIR steps are examined separately.

Fault Detection

Fault Detection requires active or passive monitoring to be carried out. Active monitoring is assigned to Built-in-Test (BIT) routines, which could be part of the operating system or embodied in utilities associated to hardware components. In the first case, the complexity of the DOS is increased, as it must provide hardware-specific test routines. In the second case the DOS, although simpler, has to properly interface the MSL to schedule BIT routines and to collect and store the tests results (fault reporting). It should be noticed how the presence of a standardized MOS should by itself solve these interface problems, recommending the second solution, which, due to a simpler DOS, would present advantages for DOS portability.

Passive monitoring is typically represented by dedicated logic incorporated in hardware devices, which reports the supervened fault event to the DOS, usually by means of simple interfaces such as interrupt lines. The DOS provides for the recording of the fault and the resetting of the hardware device. Another kind of passive monitoring, less usual, is the one implemented controlling the input / output of functional applications, in order to detect any error in their execution, and therefore deduce the fault of the processing component running that application. The DOS is in charge of recording the fault for successive fault handling steps.

Fault Identification (Diagnosis)

The identification of a fault requires the identification of the faulty component and the classification of the fault.

The identification of the source of the fail can be a very complex task to be carried out, depending on the complexity of the system and on the multiplicity of the symptoms detected. The operating system / system applications (depending on whether or not the system applications are explicited in the selected software architecture), in order to carry out this task, must have access to detailed information concerning the system configuration and the description / classification of the possible faults which can arise in the system. This information is contained in blueprints.

The complete classification of a fault requires the operating system / system applications to determine whether the fault is likely to be permanent or temporary. A temporary fault condition lasts a limited period of time and then disappears, being its presence related to temporary internal or external (environmental) conditions. A permanent fault, once appeared, remains. This determination can be done as follows: when a BIT routine detects a fault, the operating system / systems application, after having recorded the fault, continue to schedule the same routine with a suitable temporal distribution, and a decision is taken on the fault duration after a proper number of cycles. Following this decision, the BIT routine can still run in background, to identify possible restoration of a normal operational state.

Fault Recovery

After the fault has been detected and identified, proper remedial actions are carried out, that is, recovery operations commensurate with the system requirements are initiated. For sake of precision, recovery must be preceded by passivation, that is, replacement of the faulty element. This replacement can be of two types: temporal or spatial replacement.

Temporal replacement is based on retries. This is the case of most communication protocols, where detected transmission mistakes, if not recovered by means of corrector codes, require the same transmission to be repeated. No use is made of redundant physical resources, while time has to be redundant. The operating system is in charge of initiating the first retry, and, if necessary and possible, successive retries. The recovery following a temporal replacement does not need any dedicated action of the operating system / system applications, being direct consequence of the first successful retry.

Spatial replacement is based on the physical substitution of the faulty resource (module, microprocessor, link) with a non-faulty resource, and requires successive recovery actions to be carried out in order to restore correct system operation. The non-faulty spare elements are to be extracted from a pool of redundant resources within the boundary of a certain reconfiguration area. The number of spare redundant units to be allocated depends on the

extension of the reconfiguration area, on reliability/availability requirements and on the fault probability associated to each resource. The basic types of redundant resources, in modular avionics applications, can be links, switching and networking elements, microprocessors, modules (data, signal, graphic, cryptographic processing modules, mass memory modules).

Consequently to a spacial replacement, recovery procedures have to take place, whose impact on operating system / system application requirements depends on which of three different configurations methods is used for the spare resources [7] :

Hot spares: they execute, in parallel, the tasks being executed by the primary elements, which, when faulty, have to be substituted. Hot spares do not need any initalisation to become primary elements, but provide a quite static redundancy, as each hot spare is associated to a specific primary element (committed resources). This implies, for a complex system, a huge number of overhead resources. The use of hot spares has the minimum requirements on the operating system / system applications, which have to keep trace of the new system configuration, but, once replacement has been performed, required recovery actions are minimized.

Cold spares: they do not execute tasks in parallel with primary elements. When a fault takes place, they must be completely initialized before becoming active. This initialization is managed by the operating system / system applications, together with the updating of reconfiguration tables. The impact on the operating system / system applications is greater with respect to hot spares, but in this case the resources are uncommitted, and the reconfiguration is dynamic. The problem is that the initialization operations take time, and this is true in particular if the applications to be executed are not stored from the beginning in the local memory of the spare microprocessor, but have to be loaded in real-time from the global module memory or, even worse, from some other module of the reconfiguration area. In any case, the use of cold spares is likely to result in a loss of service while the initialization process takes place, which is not tolerable for real time critical applications, as avionics.

Warm spares: as cold spares, they are not statically allocated to the active elements, and do not execute code in parallel with them. Nevertheless, they do not need to be completely re-initialized before use, as they are constantly provided with periodical pictures of error-free states reached by the active elements they could be requested to substitute in case of fault. When required, they have just to step back to the proper error free state, procedure faster than a complete initialization. Warm spares do not present, therefore, the shortcomings of hot spares (static allocation of redundant resources) and cold spares (temporary loss of service), but their adoption has an higher impact on the recovery services which the operating systems / system applications must provide. The operating system / system applications should first determine, for all applications running on all system elements, when the picture of an error free state has to be taken. This implies capabilities hard to be provided, so the problem may be solved letting applications signal the operating system when a significant state takes place. The pictures (state variables, control variables, parameters of all kinds related with the execution of the relevant applications) need then to be stored at specific storage points of the system, with additional requirements on the communication management and file management services. Finally, the recovering of a normal operational state may require various error-free state pictures to be consistently restored in the pertinent locations, proportionally to the extension of the functional area affected by the fault.

An extension of the use of spatial replacement is the suppression (total or partial) of lower priority tasks on active elements, in order to make available, on these elements, enough processing power to execute higher priority tasks previously allocated on elements which have then become faulty. This procedure must be applied when previous faults have already exhausted unloaded spares, and a new fault has somehow to be tolerated. The tolerance is in this case partial, while the system performances gracefully degrade.

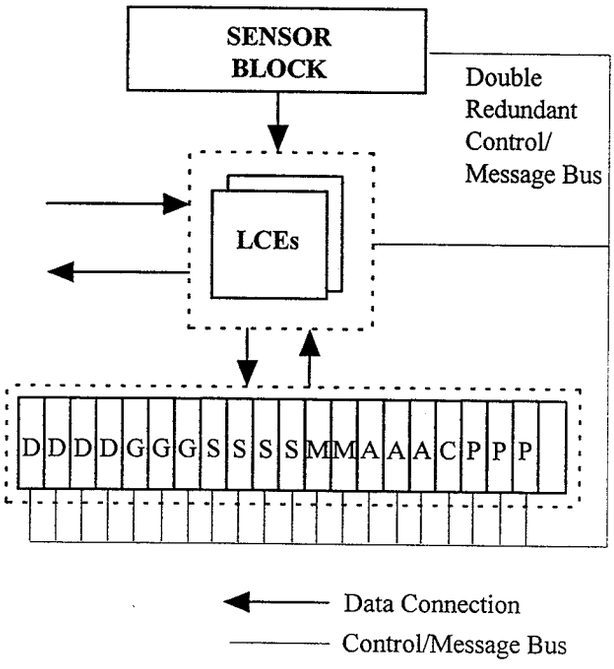
Error recovery

Errors have been defined as alteration of information units within the operational system, which origine from not recovered faults. Being the fault not recovered, the error will continue to be present in the system, nevertheless error masking methods are still applicable. With masking, due to built-in mechanisms, the error becomes transparent for the system, in principle without the intervention of the operating system. Typical example of masking mechanisms are error corrector codes such as Hamming codes or BCH (Bose-Chouduri-Hocquenghem) codes. At an higher level, masking is obtained by elements replication, that is, the traditional approach of safety critical subsystems, where multiple channels are cross-monitored and a voting system is employed to take the best decision on the output. The identification of the faulty channel can be local, the knowledge residing in a comparator unit without needing to be spread across the operating system. Of course, synchronization problems among the concurrent processing elements must be solved, being complicated by fluctuations of the sensors data. The possible configurations of the replicated units

are various, ranging from simple dual to triplex, quadruplex, dual-dual.
Voting techniques can be used to tolerate errors originating from software faults too. Software faults due to design / development errors cannot of course be tolerated by simple replication, which would replicate errors as well. N-Version programming [8] can be used, which requires n processors to concurrently execute n independent designed and developed software modules, different realization of the same application. The results are sent to a decision algorithm that delivers the optimum decision. The very low probability of similar errors makes N-version programming an effective method for achieving software fault tolerance.

A physical hypothesis

Fig. 3 [9] shows the generic reconfiguration area of the integrated modular avionics architecture outlined by EUCLID RTP4.1 research studies.



C: Cryptographic LRM
D: Data Processing LRM
M: Mass Memory LRM
G: Graphic Processing LRM
P: Power Supply LRM
A: Array Processing LRM
S: General Signal Processing LRM

LRM: Line Replaceable Module
LCE: Link Control Element

Fig. 3 - Generic Section of Modular Core Architecture

While a detailed discussion of the subject is reported in Ref [9] , the following explanatory considerations are herein sufficient:

- Different kinds of modular resources are present, such as data, signal (general and array), graphic, cryptographic processing modules, mass memory modules, power supply modules. Any kind of processing module contains in general a defined number of processors, local memories, a global memory and I/O ports.

- The generic sensor block is connected to the modular core processing resources by means of high band point-to-point optical links, configurable by means of switching elements (Link Control Elements - LCE). The modules communicate among them by means of the same kind of optical links, and of the same switching elements too.

- A secondary network is present (the Control / Message Bus) suitable to carry control / status information and lower rate data transfers. In particular, commands issued toward the LCE to reconfigure the high band optical links are dispatched by means of this secondary network.

In case the functional area supports safety critical applications, the applied fault tolerant techniques could be replication and voting. For example, the 4 data processing modules perform the same tasks, and their results are compared by a consensus algorithm delivering an agreement / disagreement decision.

In case the functional area supports mission critical applications, the technique applied to maximize reliability and availability will be FDIR with use of temporal replacement, plus spatial replacement of warm spares. Let us suppose, for example, that DPM 1,2 and 3 are the active modules, while DPM 4 is a warm spare. Local level FDIR may be applied to any active module, supposing for example that 1 of the n processors contained on the module can be used to substitute any faulty processor on that module. System level FDIR applies, for example, when an entire module is faulty, and all the steps above examined are carried out: fault detection, identification, passivation with spatial replacement and recovery by means of warm spare DPM 4.

Should a second module be faulty, a redistribution of high priority tasks, run on this module, has to take place at the expenses of lower priority tasks run on non-faulty modules (graceful degradation). In order to carry out all these recovery steps, the operating system / system applications must feature all the above capabilities.

Conclusions

The paper has presented a classification of fault handling techniques which can be utilized in modular multiprocessing avionics architecture in order to maximize system reliability and availability. The fundamental role of the operating system / system applications in providing fault tolerant operation has been highlighted. As a consequence, the operating system reliability itself turns out to be a key issue. There is an important rule to be observed to these aim: the realization of a closed environment, in which each process has exactly the capabilities and priorities requested to perform its tasks, and no more. A well structured and through error recording and reporting system, allowing engineers an effective off-line analysis of faults events and FDIR mechanisms, is a mandatory feature too.

List of references

1. Andrew L. Benjamin, D. Jaynarayan H. Lala, "Advanced Fault Tolerant Computing for Future Manned Space Missions" 16th Digital Avionics Systems Conference, Irvine, CA, 1997

2. Dal Cin, A. Grygier, H. Hessenauer, U. ildebrand, J. Hoenig, W. Hohl, E. Michel, A. Pataricza "Fault Tolerance in Distributed Shared Memory Multiprocessors", Lecture Motes in Computer Science 732, Parallel Computer Architectures.

3. W. Wilcock, "Enhanced Fault Management for Future IMA Systems", 16th Digital Avionics Systems Conference, Irvine, CA, 1997

4. A. Edwards, "ASAAC Phase 1 Harmonized Concept Summary", ERA 1994 Avionics Conference and Exhibition

5. Marchetto, "Structured definition of Modular Avionics Architectures Using Blueprints", 16th Digital Avionics Systems Conference, Irvine, CA, 1997

6. Harper, L.S. Alger, C.A. Babikyan, B.P. Butler, S.A. Friend, R.J. Ganska, J.H. Lala, T.K. Masotto, .J. Meyer, D.P. Morton, G.A. Nagle, C.E. Sakamaki, "Advanced Information Processing System: The Army Fault Tolerant Architecture Conceptual Study - Design and Analysis", NASA Contractor Report 189632,Volume II

7. Marchetto, R. Ferrato "Key Concepts in Integrated Modular Avionic Systems Architecture Design", ERA 1996 Avionics Conference and Exhibition

8. Algirdas Avizienis, "The N-Version Approach to Fault-Tolerant Software", IEEE Transactions on Software Engineering, Vol. II, No. 12, December 1985

9. Marchetto, "Modular Avionics System Architecture Definition in the EUCLID Research and Technology Programme 4.1: Methodology and Results, AGARD Conference Proceedings 581 Advanced Architectures for Aerospace Mission Systems, Istanbul, Turkey, 1996